Name: Rachel Collier

Homework 8

(400 points) **Hours:**

The homework is to be turned in before 5:30 PM of the due date class.

Also, an implementation in Visual Studio 2019 or WEBGL is required, thus you are to submit the ZIPPED project (the top Homework8 folder) to CANVAS.

I UNDERSTAND THAT TURNING ANOTHER's WORK IN is CHEATING.
I UNDERSTAND THAT ANY KIND OF DISSEMINATION of this WORK is CHEATING.

I CERTIFY THAT THE HOMEWORKs SOLUTIONs ARE MY OWN WORK!

SIGNATURE: Rachel Collier

V	X	?

HOMEWORK CHECKLIST (YOU MUST GRADE YOURSELF!):

1.	A. 200 points - please count!	200 points
2.	B. 200 points - please count!	200 points
3.	Homework8.zip NOT submitted to CANVAS?	-100 points

PLEASE ENTER YOUR GRADE IN THIS BOX: 400

Please rename Homework 8.doc to either **score**.OpenGL.doc or **score**.WEBGL.doc

NOTE: **Chapter 11 of the textbook shows the procedural methods of modeling that allow computer graphics to be used to model physical constraints and complex behavior of objects.**

Part A is Theoretical.
Part B is an OpenGL or WEBGL application.

Please note that this Homework is worth 400 points.

A. **(200 pts)**

(There are 25 questions each worth 5 to 10 points)

(Answer should be around 7 lines.)

11.1 ALGORITHMIC MODELS

a. **(10 points)** Explain.

ANSWER:

Astute researchers and application programmers have suggested that we would not require as many polygons if we could render a model generating only those polygons that are both visible and project to an area at least the size of one pixel. We have seen examples of this idea in previous chapters, for example, when we considered culling polygons before they reached the rendering pipeline. Nevertheless, a more productive approach has been to reexamine the way in which we do our modeling and seek techniques, known as procedural methods, that generate geometrical objects.

11.2 PHYSICALLY-BASED MODELS AND PARTICLE SYSTEMS

a. **(10 points)** Explain.

ANSWER:

Particle systems are collections of particles, typically point masses, in which the dynamic behavior of the particles can be determined by the solution of sets of coupled differential equations. Particle systems have been used to generate a wide variety of behaviors in a number of fields. In fluid dynamics, people use particle systems to model turbulent behavior. Rather than solving partial differential equations, we can simulate the behavior of the system by following a group of particles that is subject to a variety of forces and constraints.

11.3 NEWTONIAN PARTICLES

a. **(10 points)** Explain.

ANSWER:

The dynamic state of the system is obtained by numerical methods that in- volve stepping through approximations to the set of differential equations. A typical time step is based on computing the forces that apply to the n particles through a user-defined function, using these forces to update the state through a numerical differential-equation solver, and finally using the new positions of the particles and their attributes to render whatever graphical objects we wish to place at the particles' locations.

11.3.1 Independent PARTICLES

a. **(10 points)** Explain.

ANSWER:

There are numerous simple ways that particles can interact and determine the forces that act on each particle. If the forces that act on a given particle are independent of other particles, the force on the ith particle can be described by the equation
fi = fi(pi, vi). A simple case occurs where each particle is subject only to a constant gravitational force fi/mi = g. If this force points down, then where g is positive, and each particle will trace out a parabolic arc. If we add a term proportional to the velocity, we can have the particle subject to frictional forces, such as drag.

11.3.2 Spring Forces

a. **(10 points)** Explain.
ANSWER
One method to model the forces among particles is to consider adjacent parti- cles as connected by a spring. Consider two adjacent particles, located at p and q, connected by a spring, as shown in Figure 11.2. Let f denote the force acting on p from q. A force −f acts on q from p. The spring has a resting length s, which is the distance between particles if the system is not subject to external forces and is allowed to come to rest. When the spring is stretched, the force acts in the direction d = p − q; that is, it acts along the line between the points.

11.3.3 Attractive and Repulsive Forces

a. **(10 points)** Explain.
ANSWER
Whereas spring forces are used to keep a group of particles together, repulsive forces push particles away from one another and attractive forces pull particles toward one another. We could use repulsive forces to distribute particles over a surface, or if the particles represent locations of objects, to keep objects from hitting one another. We could use attractive forces to build a model of the solar system or to create applications that model satellites revolving about the earth. The equations for attraction and repulsion are essentially the same except for a sign.

11.4 SOLVING PARTICLE SYSTEMS

a. **(10 points)** Explain.
ANSWER:
There are two potential problems with Euler's method: accuracy and stability. Both are affected by the step size. The accuracy of Euler's method is proportional to the square of the step size. Consequently, to increase the accuracy we must cut the step size, thus increasing the time it takes to solve the system. A potentially more serious problem concerns stability. As we go from step to step, the per-step errors that we make come from two sources: the approximation error that we made by using the Taylor series approximation and the numerical errors that we make in computing the functions.

11.5 CONSTRAINTS

a. **(10 points)** Explain
ANSWER:
There are two types of constraints that we can impose on particles. Hard constraints are those that must be adhered to exactly. For example, a ball must bounce off a wall; it cannot penetrate the wall and emerge from the other side. Nor can we allow the ball just to come close and then go off in another direction. Soft constraints are those that we need only come close to satisfying. For example, we might want two particles to be separated approximately by a specified distance, as in a particle mesh.

11.5.1 Collisions

a. **(10 points)** Explain
ANSWER:
Consider the problem of collisions. We can separate the problem into two parts: detection

and reaction. Suppose that we have a collection of particles and other geometric objects and the particles repel one another. We therefore need to consider only collisions between each particle and the other objects. If there are n particles and m polygons that define the geometric objects, at each time step we can check whether any particle has gone through any of the polygons. We can detect this collision by inserting the position of the particle into the equation of the plane of the polygon.

11.5.2 Soft Constraints

a. **(10 points)** Explain

ANSWER:

Most hard constraints are difficult to enforce. For example, if we want to ensure that a particle's velocity is less than a maximum velocity or that all the particles have a constant amount of energy, then the resulting mathematics is far more difficult than what we have already seen, and such constraints do not always lead to a simple set of ordinary differential equations. In many situations, we can work with soft constraints: constraints that we need to come only close to satisfying.

11.6 A SIMPLE PARTIAL SYSTEM

a. **(10 points)** Explain.

ANSWER:

We conclude our development of partial systems by building a simple particle system that can be expanded to more complex behaviors. Our particles are all Newtonian so their state is described by their positions and velocities. In addition, each particle can have its own color and mass. A particle system is an array of particles.

11.6.1 Displaying the Particles

a. **(5 points)** Explain.

ANSWER:

Given the position of a particle, we can display it using any set of primitives, either geometric or raster, that we like. A simple starting point would be to display each particle as a point. The colors are stored in an array of colors as follows.

```
GLfloat colors[8][3]={{0.0, 0.0, 0.0}, {1.0, 0.0, 0.0}, {0.0, 1.0, 0.0},
   {0.0, 0.0, 1.0}, {0.0, 1.0, 1.0}, {1.0, 0.0, 1.0}, {1.0, 1.0, 0.0},
   {1.0, 1.0, 1.0}};
```

11.6.2 Updating the Particle Positions

a. **(5 points)** Explain.

ANSWER:

The positions are updated using the velocity and the velocity is updated by computing the forces on that particle. We have assumed that the time interval is short enough that we can compute the forces on each particle as we update its state. A more robust strategy would be to compute all the forces on all the particles first and put the results into an array that can be used to update the state. We will use the collision function to keep the particles inside a box. It could also be used to deal with collisions between particles.

11.6.3 Initialization

a. **(5 points)** Explain.

ANSWER:

We start by creating a default number of particles that we place randomly within a cube centered at the origin with sides of length 2 and assign each a random velocity. The initial number of particles and default speed are specified in define statements but can be altered through a menu.

11.6.4 Collisions

a. **(5 points)** Explain.

ANSWER:

We use the collision function to keep the particles inside the initial axis-aligned box. Our strategy is to increment the position of each particle and then check to see if the particle has crossed one of the sides of the box. If it has crossed a side then we can treat the bounce as a reflection. Thus, we need only to change the sign of the velocity in the normal direction and place the particle on the other side of the box. If the coefficient of restitution is less than 1.0, the particles will slow down when they hit a side of the box.

11.6.5 Forces

a. **(5 points)** Explain.

ANSWER:

If the forces are set to zero, the particles will bounce around the box on linear paths continuously. If the coefficient is less than 1.0, eventually the particles will slow to a halt. The easiest force to add is gravity. For example, if all the particles have the same mass, we can add a gravitational term in the y-direction as follows. We can add repulsive or attractive forces by computing the distances between all pairs of particles at the beginning of each iteration and then using any inverse square term.

11.6.6 Flocking

a. **(5 points)** Explain.

ANSWER:

Some of the most interesting applications of particle system are for simulating com- plex behaviors among the particles. Perhaps a more accurate statement is that we can produce what appears to be complex behavior using some simple rules for how par- ticles interact. A classic example is simulating the flocking behavior of birds. How does a large group of birds maintain a flock without each bird knowing the positions of all the other birds? We can investigate some possibilities by making some minor modifications to our particle system.

11.7 LANGUAGE-BASED MODELS

a. **(10 points)** Explain.

ANSWER:

If we look at only the syntax of a language, there is a direct correlation between the rules of the language and the form of the trees that represent the sentences. We can extend this idea to hierarchical objects in graphics, relating a set of rules and a tree- structured model. These systems are known as tree grammars. A grammar can be defined by a set of symbols and a set symbol-replacement rules, or productions, that specify how to replace a symbol by one or more symbols.

11.8 RECURSIVE METHODS AND FRACTALS

a. **(10 points)** Explain.

ANSWER:

The language-based procedural models offer but one approach to generating com- plex objects with simple programs. Another approach, based on fractal geometry, uses the self-similarity of many real-world objects. Fractal geometry was developed by Mandelbrot, who was able to create a branch of mathematics that enables us to work with interesting phenomena that we cannot describe using the tools of ordinary geometry. Workers in computer graphics have used the ideas of fractal geometry not only to create beautiful and complex objects, but also to model many real-world entities that are not modeled easily by other methods.

11.8.1 Rulers and Length

a. **(5 points)** Explain.

ANSWER

There are two pillars to fractal geometry: the dependence of geometry on scale and self-similarity. We can examine both through the exploration of one of the questions that led to fractal geometry: What is the length of a coastline? Suppose that we have a map of a coastline. Because the coastline is wavy and irregular, we can take a string, lay it over the image of the coastline, and then measure the length of the string, using the scale of the map to convert distances. However, if we get a second map that shows a closer view of the coastline, we see more detail.

11.8.2 Fractal Dimensions

a. **(5 points)** Explain.

ANSWER

We divide the line segment into k = n identical segments, the square into k = n2 small squares, and the cube into k = n3 small cubes. In each case, we can say that we have created new objects by scaling the original object by a factor of h and replicating it k times. Suppose that d is the dimension of any one of these objects. What has remained constant in the subdivision is that the whole is the sum of the parts. Mathematically, for any of the objects, we have equality.

11.8.3 Midpoint Division and Brownian Motion

a. **(5 points)** Explain.

ANSWER

In computer graphics, there are many situations where we would like to create a curve or surface that appears random but has a measurable amount of roughness. For example, the silhouette of a mountain range forms a curve that is rougher (has higher fractal dimension) than the skyline of the desert. Likewise, a surface model of mountain terrain should have a higher fractal dimension than the surface of farmland. We also often want to generate these objects in a resolution-dependent manner.

11.8.4 Fractal Mountains

a. **(5 points)** Explain.

ANSWER

The best-known uses of fractals in computer graphics have been to generate mountains and terrain. We can generate a mountain with our tetrahedron-subdivision process, by adding in a midpoint displacement. Consider one facet of the tetrahedron. First, we find the midpoints of the sides; then we displace each midpoint, creating four new triangles. Once more, by controlling the variance of the random-number generator, we can control the roughness of the resulting object.

11.8.5 The Mandelbrot Set

a. **(10 points)** Explain.

ANSWER

The famous Mandelbrot set is an interesting example of fractal geometry that can be generated easily with OpenGL's pixel drawing functionality. Although the Mandelbrot set is easy to generate, it shows infinite complexity in the patterns it generates. It also provides a good example of generating images and using color lookup tables. In this discussion, we assume that you have a basic familiarity with complex arithmetic. We denote a point in the complex plane as $z = x + iy$, where x is the real part and y is the imaginary part of z.

11.9 PROCEDURAL NOISE

a. **(10 points)** Explain.

ANSWER:

We have used pseudorandom-number generators to generate the Sierpinski gasket and for fractal subdivision. The use of pseudorandom-number generators has many other uses in computer graphics ranging from generating textures to generating mod- els of natural objects such as clouds and fluids. However, there are both practical and theoretical reasons that the simple random-number generator that we have used is not a good choice for our applications. Within the computer we can generate pseudorandom sequences of numbers. These random-number generators, such as the function rand that we have used, produce uncorrelated sequences, but because the sequences repeat after a long period, they are not truly random.

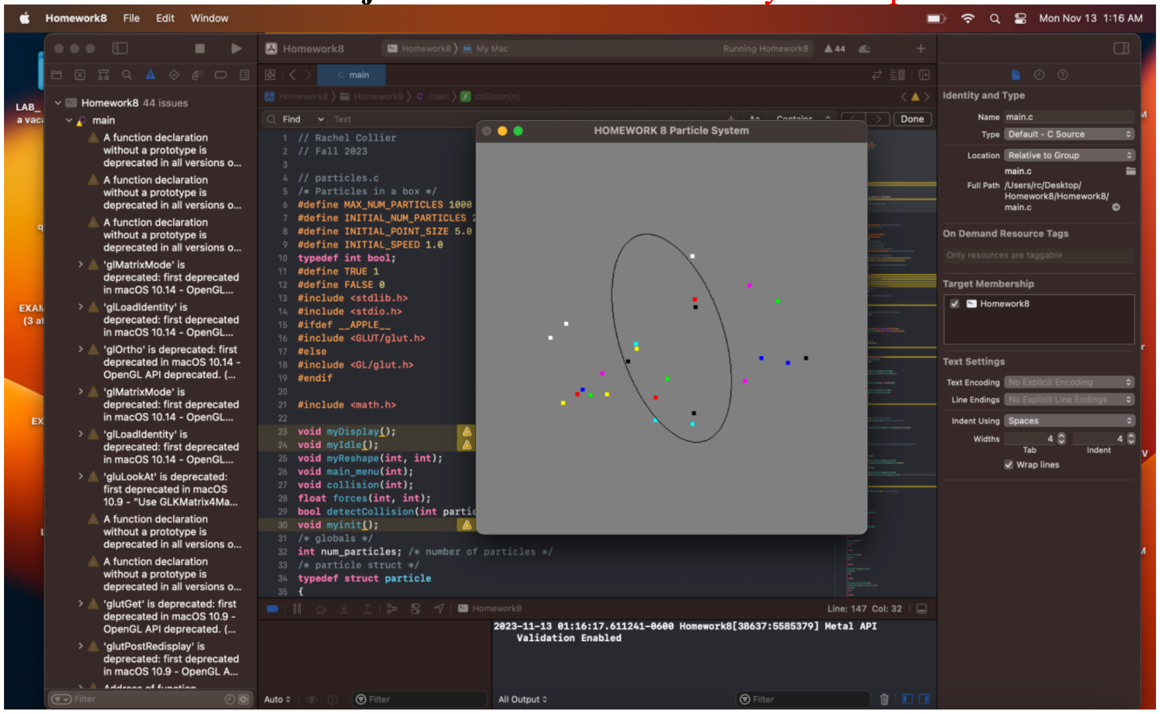
B. (200 pts) Visual Studio 2019 C++ Project

B1. **(150 points)**
Create Visual Studio 2019 C++, Empty Project, Homework8:

Modify the particles.c from Lecture 11:

1. `glutCreateWindow("HOMEWORK 8 Particle System");`
2. Instead of particles in a "box" modify the program for particles in a "sphere".
3. Add code that when the particles colide with each other an explosion or fireworks effect will take place.

Build and run this Project: Insert screenshots of your output.



A function declaration without a prototype is deprecated in all versions o...
A function declaration without a prototype is deprecated in all versions o...
A function declaration without a prototype is deprecated in all versions o...
'glMatrixMode' is deprecated: first deprecated in macOS 10.14 - OpenGL...
'glLoadIdentity' is deprecated: first deprecated in macOS 10.14 - OpenGL...
'glOrtho' is deprecated: first deprecated in macOS 10.14 - OpenGL API deprecated. (...
'glMatrixMode' is deprecated: first deprecated in macOS 10.14 - OpenGL...
'glLoadIdentity' is deprecated: first deprecated in macOS 10.14 - OpenGL...
'gluLookAt' is deprecated: first deprecated in macOS 10.9 - "Use GLKMatrix4Ma...
A function declaration without a prototype is deprecated in all versions o...
A function declaration without a prototype is deprecated in all versions o...
'glutGet' is deprecated: first deprecated in macOS 10.9 - OpenGL API deprecated. (...
'glutPostRedisplay' is deprecated: first deprecated in macOS 10.9 - OpenGL A...

```c
// Rachel Collier
// Fall 2023

// particles.c
/* Particles in a box */
#define MAX_NUM_PARTICLES 1000
#define INITIAL_NUM_PARTICLES 2
#define INITIAL_POINT_SIZE 5.0
#define INITIAL_SPEED 1.0
typedef int bool;
#define TRUE 1
#define FALSE 0
#include <stdlib.h>
#include <stdio.h>
#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

#include <math.h>

void myDisplay();
void myIdle();
void myReshape(int, int);
void main_menu(int);
void collision(int);
float forces(int, int);
bool detectCollision(int partic
void myinit();
/* globals */
int num_particles; /* number of particles */
/* particle struct */
typedef struct particle
{
```

```
2023-11-13 01:16:17.611241-0600 Homework8[38637:5585379] Metal API
    Validation Enabled
```

B2. **(50 points)**
Add your code below (must have your name and semester at the top):

ANSWER:

```c
// Rachel Collier
// Fall 2023
// particles.c
/* Particles in a box */
#define MAX_NUM_PARTICLES 1000
#define INITIAL_NUM_PARTICLES 25
```

```
#define INITIAL_POINT_SIZE 5.0
#define INITIAL_SPEED 1.0
typedef int bool;
#define TRUE 1
#define FALSE 0
#include <stdlib.h>
#include <stdio.h>
#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif
#include <math.h>
void myDisplay();
void myIdle();
void myReshape(int, int);
void main_menu(int);
void collision(int);
float forces(int, int);
bool detectCollision(int particle1, int particle2);
void myinit();
/* globals */
int num_particles; /* number of particles */
/* particle struct */
typedef struct particle
{
int color;
float position[3];
float velocity[3];
float mass;
} particle;
particle particles[MAX_NUM_PARTICLES]; /* particle system */
/* initial state of particle system */
int present_time;
int last_time;
int num_particles = INITIAL_NUM_PARTICLES;
float point_size = INITIAL_POINT_SIZE;
float speed = INITIAL_SPEED;
bool gravity = FALSE; /* gravity off */
bool elastic = FALSE; /* restitution off */
bool repulsion = FALSE; /* repulsion off */
float coef = 1.0; /* perfectly elastic collisions */
float d2[MAX_NUM_PARTICLES][MAX_NUM_PARTICLES]; /* array for interparticle
distances */
GLsizei wh = 500, ww = 500; /* initial window size */
GLfloat colors[8][3]={{0.0, 0.0, 0.0}, {1.0, 0.0, 0.0},{0.0, 1.0, 0.0},
{0.0, 0.0, 1.0}, {0.0, 1.0, 1.0}, {1.0, 0.0, 1.0}, {1.0, 1.0, 0.0},
{1.0, 1.0, 1.0}};
/* rehaping routine called whenever window is resized or moved */
void myReshape(int w, int h)
{
/* adjust clipping box */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

```
glOrtho(-2.0, 2.0, -2.0, 2.0, -4.0, 4.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(1.5,1.0,1.0,0.0,0.0,0.0,0.0,1.0,0.0);
/* adjust viewport and clear */
if(w<h) glViewport(0,0,w,w);
else glViewport(0,0,h,h);
/* set global size for use by drawing routine */
ww = w;
wh = h;
}
void myinit()
{
int i, j;
/* set up particles with random locations and velocities */
for(i=0; i<num_particles; i++)
{
particles[i].mass = 1.0;
particles[i].color = i%8;
for(j=0; j<3; j++)
{
particles[i].position[j] = 2.0*((float) rand()/RAND_MAX)-1.0;
particles[i].velocity[j] = speed*2.0*((float) rand()/RAND_MAX)-1.0;
}
}
glPointSize(point_size);
/* set clear color to grey */
glClearColor(0.5, 0.5, 0.5, 1.0);
}
void myIdle()
{
int i, j, k;
float dt;
present_time = glutGet(GLUT_ELAPSED_TIME);
dt = 0.001*(present_time - last_time);
for(i=0; i<num_particles; i++)
{
for(j=0; j<3; j++)
{
particles[i].position[j]+=dt*particles[i].velocity[j];
particles[i].velocity[j]+=dt*forces(i,j)/particles[i].mass;
}
collision(i);
}
if(repulsion) for(i=0;i<num_particles;i++) for(k=0;k<i;k++)
{
d2[i][k] = 0.0;
for(j=0;j<3;j++) d2[i][k]+= (particles[i].position[j]-
particles[k].position[j])*(particles[i].position[j]-
particles[k].position[j]);
d2[k][i]=d2[i][k];
}
last_time = present_time;
glutPostRedisplay();
```

```
}
float forces(int i, int j)
{
int k;
float force = 0.0;
if(gravity&&j==1) force = -1.0; /* simple gravity */
if(repulsion) for(k=0; k<num_particles; k++) /* repulsive force */
{
if(k!=i) force+=
0.001*(particles[i].position[j]-particles[k].position[j])/(0.001+d2[i][k]);
}
return(force);
}
void collision(int n)
/* tests for collisions against cube and reflect particles if necessary */
{
int i;
for (i=0; i<3; i++)
{
if(particles[n].position[i]>=1.0)
{
particles[n].velocity[i] = -coef*particles[n].velocity[i];
particles[n].position[i] = 1.0-coef*(particles[n].position[i]-1.0);
}
if(particles[n].position[i]<=-1.0)
{
particles[n].velocity[i] = -coef*particles[n].velocity[i];
particles[n].position[i] = -1.0-coef*(particles[n].position[i]
+1.0);
}
}

   if (detectCollision)
         {
            particles[n].mass *= 67.5; // Increase size
         }
}
void main_menu(int index)
{
switch(index)
{
case(1):
{
num_particles = 2*num_particles;
myinit();
break;
}
case(2):
{
num_particles = num_particles/2;
myinit();
break;
}
case(3):
```

```
{
speed = 2.0*speed;
myinit();
break;
}
case(4):
{
speed = speed/2.0;
myinit();
break;
}
case(5):
{
point_size = 2.0*point_size;
myinit();
break;
}
case(6):
{
point_size = point_size/2.0;
if(point_size<1.0) point_size = 1.0;
myinit();
break;
}
case(7):
{
gravity = !gravity;
myinit();
break;
}
case(8):
{
elastic = !elastic;
if(elastic) coef = 0.9;
else coef = 1.0;
myinit();
break;
}
case(9):
{
repulsion = !repulsion;
myinit();
break;
}
case(10):
{
exit(0);
break;
}
}
}
void myDisplay()
{
   int i;
```

```
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POINTS); /* render all particles */
    for(i=0; i<num_particles; i++)
    {
        glColor3fv(colors[particles[i].color]);
        glVertex3fv(particles[i].position);
    }
    glEnd();
    glColor3f(0.0,0.0,0.0);

    // Draw the outline of the circle
    glBegin(GL_LINE_LOOP);
    for (i = 0; i < 360; i++)
    {
        float theta = i * 3.14159265358979323846 / 180.0;
        float x = cos(theta) * 1.1; // Radius of 1.1 for the circle
        float y = sin(theta) * 1.1;
        glVertex3f(x, y, 0.0);
    }
    glEnd();

    glutSwapBuffers();
}
int main(int argc, char** argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutCreateWindow("HOMEWORK 8 Particle System");
    glutDisplayFunc(myDisplay);
    myinit ();
    glutCreateMenu(main_menu);
    glutAddMenuEntry("more particles", 1);
    glutAddMenuEntry("fewer particles", 2);
    glutAddMenuEntry("faster", 3);
    glutAddMenuEntry("slower", 4);
    glutAddMenuEntry("larger particles", 5);
    glutAddMenuEntry("smaller particles", 6);
    glutAddMenuEntry("toggle gravity",7);
    glutAddMenuEntry("toggle restitution",8);
    glutAddMenuEntry("toggle repulsion",9);
    glutAddMenuEntry("quit",10);
    glutAttachMenu(GLUT_MIDDLE_BUTTON);
    glutIdleFunc(myIdle);
    glutReshapeFunc (myReshape);
    glutMainLoop();
}
```